Node.js 事件循环的工作流程 & 生命周期

首先说说一些常见的误解

在 JS 引擎内部的事件循环

最常见的误解之一，事件循环是 Javascript 引擎（V8，spiderMonkey 等）的一部分。事实上事件循环主要利用 Javascript 引擎来执行代码。

有一个栈或者队列

首先没有栈，其次这个过程是复杂的，有多个队列（像数据结构中的队列）参与。但是大多数开发者觉得所有的回调函数是被推进一个单一的队列里面，这种想法是完全错误的。

事件循环运行在一个单独的线程里面

由于错误的 nodejs 事件循环图，我们中有一部分（早期我也是其中一员）认为有两个线程。一个执行 Javascript，另一个执行事件循环。事实上都在一个线程里面执行。

在 setTimeout 中有异步的 OS 的系统参与

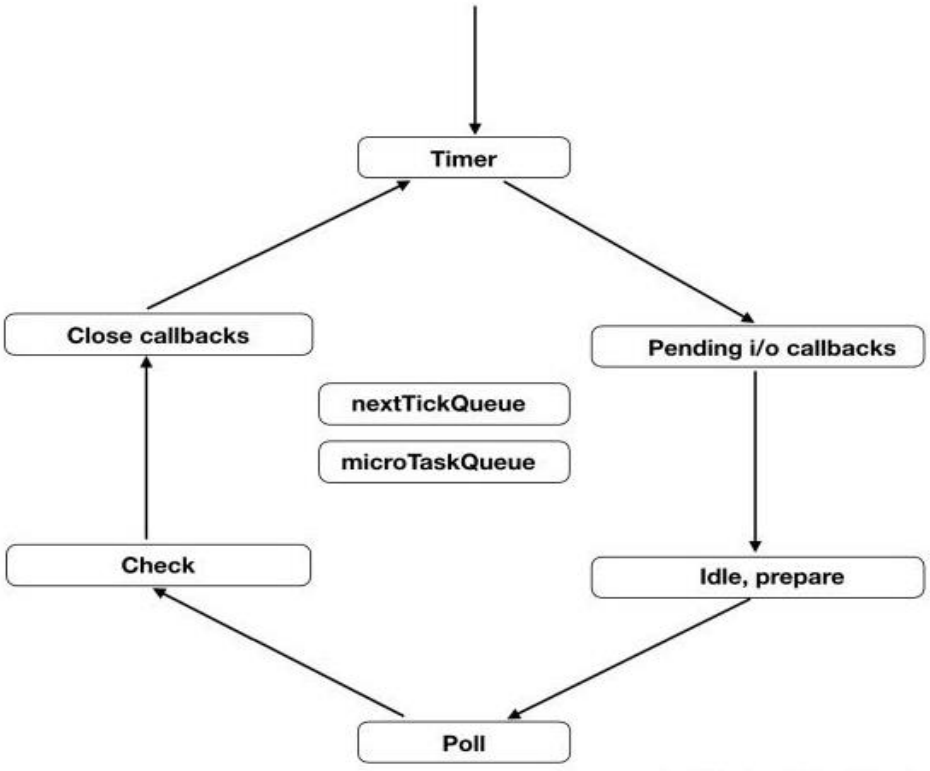
另一个非常大的误解是 setTimeout 的回调函数在给定的延迟完成之后被（可能是 OS 或者内核）推进一个队列。接下来我们会讨论这一机制。

setImmediate 将回调函数放在第一个位置

基于常见的看法 - 事件循环只有一个队列，所以一些开发者认为 setImmediate 将回调放在工作队列的前面。这是完全错误的，在 Javascript 中的工作队列都是先进先出的。

事件循环的架构

在我们开始描述事件循环的工作流程时，知道它的架构非常重要。，下面这个才是事件循环真正的工作流程。



图片中的不同的盒子代表不同的阶段，每个阶段执行特定的工作。每个阶段都有一个队列（这里说成队列主要是为了你更好的理解，真实上和数据结构种的队列有区别），Javascript 可以在任何一个阶段执行（除了 idle & prepare）。你在图片中也能看到 nextTickQueue 和 microTaskQueue，它们不是循环的一部分，它们的回调可以在任意阶段执行。它们有更高的优先级去执行。nextTickQueue 的优先级高于 microTaskQueue。

现在你知道了事件循环是不同阶段和不同队列的结合；

下面是每个阶段的描述。

定时器（Timer）阶段

这个是事件循环开始的阶段，绑定到这个阶段的队列，保留着定时器（setTimeout, setInterval）的回调。尽管它并没有将回调推入队列中，但是用最小堆来存储计时器并且在到达规定的时间后执行回调。

即将发生的（Pending） i/o 回调阶段

这个阶段执行在事件循环中 pending_queue 里的回调。这些回调是被之前的操作推入的。例如当你尝试往 tcp 中写入一些东西，这个工作完成了，然后回调被推入到队列中。错误处理的回调也在这里。

Idle, Prepare 阶段

尽管名字是空闲（idle）,但是每个循环（tick）都运行。Prepare 也在轮询阶段开始之前运行。不管怎样，这两个阶段是 node 主要做一些内部操作的阶段；因此，我们不在这儿讨论。

轮询（Poll）阶段

可能整个事件循环最重要的一个阶段就是 poll phase。这个阶段接受新传入的连接（新的 Socket 建立等）和数据（文件读取等）。我们可以将轮询阶段分成几个不同的部分。

- 如果 watch_queue（这个队列被绑定到轮询阶段）不为空，它们将会被一个接着一个的执行直到队列为空或者系统到达最大的限制。
- 一旦队列为空，node 就会等待新的连接。等待或者睡眠的时间取决于多种因素，待会儿我们会讨论。

检查（Check）阶段

轮询的下一个阶段是 check phase，这个专用于 setImmediate 的阶段。为什么需要一个专门的队列来处理 setImmediate 回调。这是因为轮询阶段的行为，待会儿将在流程部分讨论。现在只需要记住检查（check）阶段主要用于处理 setImmediate() 的回调。

关闭（Close）回调

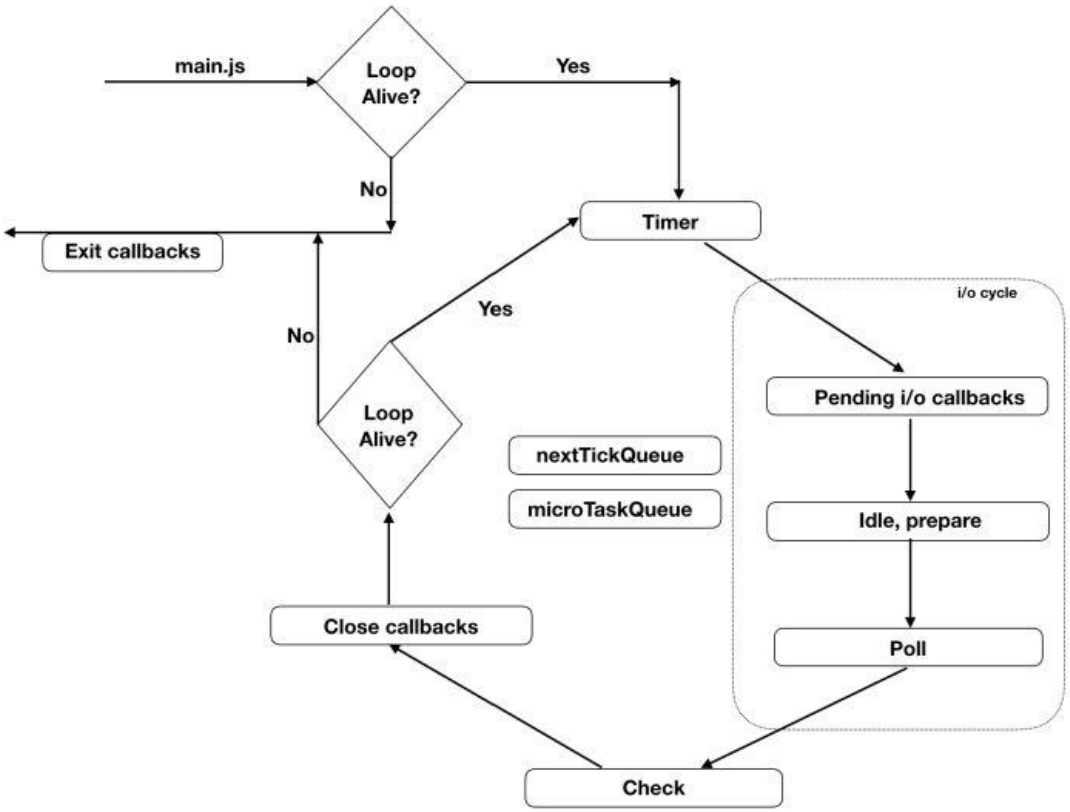
回调的关闭(socket.on(‘close’, ()=>{})) 都在这里处理的，更像一个清理阶段

nextTickQueue & microTaskQueue

nextTickQueue 中的存储着被 process.nextTick() 触发的回调。microTaskQueue 保留着被 Promise 触发的回调。它们都不是事件循环的一部分（不是在 libUV 中开发的），而是在 node.js 中。在 C/C++ 和 Javascript 有交叉的时候，它们都是尽可能快地被调用。因此它们应该在当前操作运行后（不一定是当前 js 回调执行完）。

事件循环的工作流程

当在你的控制台运行 node my-script.js，node 设置事件循环然后运行你主要的模块（my-script.js）*事件循环的外部*。一旦主要模块执行完，node 将会检查循环是否还活着（事件循环中是否还有事情要做）？如果没有，将会在执行退出回调后退出。process.on('exit', foo) 回调（退出回调）。但是如果循环还活着，node 将会从计时器阶段进入循环。



计时器阶段（Timer phase）的工作流程

事件循环进入计时器阶段并且检查在计时器队列中是否有需要执行的。好吧，这句话听起来非常简单，但是事件循环实际上要执行一些步骤发现合适的回调。实际上计时器脚本以升序存储在堆内存中。它首先获取到一个执行计器，计算下是否 now - registeredTime >= delta？如果是，他会执这个行计时器的回调并且检查下一个计时器。直到找到一个还没到约定时间的计时器， 它会停止检查其他的定时器（因为定时器都以升序排好了）并且移到下一个阶段了。

假设你调用了 setTimeout 4 次创建了 4 个定时器，分别相对于时间 t 来说 100, 200, 300, 400 的差值。

```
 _____       _____       _____       _____
| A | --> | B | --> | C | --> | D |
|___|     |___|     |___|     |___|
```

假设事件循环在 t+250 进入到了计时器阶段。它会首先看下计时器 A，A 的过期时间是 t+100。但是现在时间是 t+250。因此它将执行绑定在计时器 A 上的回调。然后去检查计时器 B，发现它的过期时间是 t+200，因此也会执行 B 的回调。现在它会检查 C，发现它的过期时间是 t+300，因此将离开它。事件循环不会去检查 D ，因为计时器都是按升序排好的；因此 D 的阈值比 C 大。然而这个阶段有一个系统相关的硬限制，如果达到系统依赖最大限制数量，

即使有未执行的计时器，它也会移到下一阶段。

即将发生的（Pengding phase）的 i/o 阶段工作流程

计时器阶段后，事件循环将会进入到了即将发生的 i/o 阶段，然后检查一下 pengding_queue 中是否有来自于之前的即将发生的任务的回调。如果有，一个接一个的执行，直到队列为空，或者达到系统的最大限制。之后，事件循环将会移到 idle handler 阶段，其次是准备阶段做一些内部的操作。然后最终很可能进入到了最重要的阶段 poll phase。

轮询阶段（Poll phase）工作流程

像名字说的那样，这是一个观察的阶段。观察是否有新的请求或者连接传入。当事件循环进入轮询阶段，它会在 watcher_queue 中执行脚本，包含文件读响应，新的 socket 或者 http 连接请求，直到时间耗尽或者像其他阶段那样达到系统依赖上限。假设没有要执行的回调，轮询在某些特定的条件下将会等待一会儿。如果在检查队列（check queue），即将发生的队列 pending queue），或者关闭队列（closing callbacks queue 或者 idle handler queue）里面有任何任务在等待，它将等待 0 毫秒。然而它会根据定时器堆来决定等待时间执行第一个定时器（如果可获取）。如果第一个定时器阈值经过了，毫无疑问它不需要等待（就会执行第一个定时器）。

检查阶段（Check phase）工作流程

轮询阶段结束之后，立即来到检查阶段。这个阶段的队列中有被 setImmediate 触发的回调。它将会像其他阶段那样一个接着一个的执行，直到队列为空或者达到依赖系统的最大限制。

关闭回调（Close callback）的工作流程

完成在检查阶段的任务之后，事件循环的下一个目的地是处理关闭或者销毁类型的回调 close callback。事件循环执行完这个阶段的队列中的回调后，它会检查循环（loop）是否还活着，如果没有，退出。但是如果还有工作要做，它会进行下一个循环；因此在计时器阶段。如果你认为之前例子中的定时器 （A & B）过期，那么现在在定时器阶段将会从定时器 C 开始检查是否过期。

nextTickQueue & microTaskQueue

因此，这两个队列的回调函数什么时候运行？它们当然在从当前阶段到下一个阶段之前尽可能快的运行。不像其他阶段，它们两个没有系统依赖的最大限制，node 运行它们直到两个队列是空的。然而，nextTickQueue 会比 microTaskQueue 有着更高的任务优先级。

进程池（Thread-pool）

我从 JavaScript 开发者那里听到普遍的一个词就是 ThreadPool。一个普遍的误解是，nodejs 有一个处理所有异步操作的进程池。但是实际上进程池是 libUV（nodejs 用来处理异步的第三方库） 库中的。我没有在图中画出来，因为他不是循环机制的一部分。我们可以在另一篇你文章里讲讲 libUV。目前，我只能告诉你并不是每个异步任务被进程池所处理的。libUv 能够灵活地使用操作系统的异步 apis 来保持环境为事件驱动。然而操作系统的 api 不能做文件读取，dns 查询等，这些由进程池所处理，默认只有 4 个进程。你可以通过设置 uv_threadpool_size 地环境变量增加进程地数量直到 128。

带有示例的工作流程

希望你能理解事件循环是如何工作的。C 语言中同步的 while 帮助 Javascript 成为异步的。每次只处理一件事但是很难阻塞。当然，无论我们如何描述理论，我相信最好的理解就是示例。因此，让我们通过一些代码片段来理解这个脚本。

片段 1 – 基础理解

```
setTimeout(() => {
    console.log('setTimeout');
}, 0);
```

```
setImmediate(() => {
  console.log('setImmediate');
});
```

你能够猜到上面的输出吗？好吧，你可能认为 setTimeout 会先被打印出来，但是不能保证，为什么呢？执行完主模块之后进入计时器阶段，它可能不会或者会发现你的计时器的等待时间耗尽了。为什么呢？一个计时器脚本根据系统时间和你提供的增量时间注册的。setTimeout 调用的同时，计时器脚本被写入到了内存中，根据你的机器性能和其他运行在它上面的操作（不是 node）的不同，可能会有一个很小的延迟。另一点是，node 仅仅在进入计时器阶段（每一轮遍历）之前设置一个变量 now，将 now 作为当前时间。因此你可以说相对于精确的时间有点儿问题。这就是不确定性的原因。如果你在一个计时器代码的回调里面（比如： setTimeout）执行相同的代码会得到相同的结果。

然而，如果你移动这段代码到 i/o 周期里，保证 setImmediate 回调会先于 setTimeout 运行。

```
fs.readFile('my-file-path.txt', () => {
  setTimeout(() => {
    console.log('setTimeout');
  }, 0);
  setImmediate(() => {
    console.log('setImmediate');
  });
});
```

片段 2 – 更好的理解计时器

```
var i = 0;
var start = new Date();
function foo () {
    i++;
    if (i < 1000) {
        setImmediate(foo);
    } else {
        var end = new Date();
        console.log("Execution time: ", (end - start));
    }
}
foo();
```

上面的例子非常简单。调用函数 foo 函数内部再通过 setImmediate 递归调用 foo 直到 1000。在我的 macbook pro 上面，node 版本是 8.9.1，花费了 6 到 8 毫秒。现在修改下上面的代码，把 setImmediate(foo) 换成 setTimeout(foo, 0)。

```
var i = 0;
var start = new Date();
function foo () {
    i++;
    if (i < 1000) {
        setTimeout(foo, 0);
```

```
    } else {
        var end = new Date();
        console.log("Execution time: ", (end - start));
    }
}
foo();
```

现在在我的电脑上面运行这段代码花费了 1400+ms。为什么会这样？它们都没有 i/o 事件，应该一样才对。上面两个例子等待时间是 0。为什么花费了这么长时间？通过时间比较找到了偏差，CPU 密集型任务，花费更多的时间。注册计时器脚本也花费时间。定时器的每个阶段都需要做一些操作来决定一个计时器是否应该执行。长时间的执行也会导致更多的 ticks。然而，在 setImmediate 中，没有检查这一阶段，就好像在一个队列里面然后执行就行了。

片段 3 – 理解 nextTick() & 计时器（timer） 执行

```
var i = 0;
function foo(){
  i++;
  if(i>20){
    return;
  }
  console.log("foo");
  setTimeout(()=>{
    console.log("setTimeout");
  },0);
  process.nextTick(foo);
}
setTimeout(foo, 2);
```

你认为上面的输出是什么？是的，它会输出 foo 然后输出 setTimeout。2 秒后，被 nextTickQueue 递归调用的 foo() 打印出第一个 foo。当所有的 nextTickQueue 执行完了，开始执行其他（比如 setTimeout 回调）的。

所以是每个回调执行完之后，开始检查 nextTickQueue 的吗？我们改下代码看下。

```
var i = 0;
function foo(){
  i++;
  if(i>20){
    return;
  }
  console.log("foo", i);
  setTimeout(()=>{
    console.log("setTimeout", i);
  },0);
  process.nextTick(foo);
}

setTimeout(foo, 2);
setTimeout(()=>{
```

```
  console.log("Other setTimeout");
},2);
```

在 setTimeout 之后，我仅仅用一样的延迟时间添加了另一个输出 other setTimeout 的 setTimeout。尽管不能保证，但是有可能会在输出第一个 foo 之后输出 other setTimeout 。相同的计时器分成一个组，nextTickQueue 会在正在进行中的回调组执行完之后执行。

一些普遍的问题

JavaScript 代码在哪里执行的?

就像我们大多数人都认为事件循环是在一个单独的线程里里面，将回调推入一个队列，然后一个接一个的执行。第一次读到这篇文章的读者可能会感到疑惑，JavaScript 在哪里执行的。正如我早些时候说的，只有一个线程，来自于本身使用 V8 或者其他引擎的事件循环的 Javascript 代码也是在这里运行的。执行是同步的，如果当前的 JavaScript 执行还没有完成，事件循环不会传播。

我们有了 setTimeout(fn, 0)，为什么还需要 setImmediate?

首先不是 0，而是 1。当你设置一个计时器，时间为小于 1，或者大于 2147483647ms 的时候，它会自动设置为 1。因此你如果设置 setTimeout 的延迟时间为 0，它会自动设置为 1。*译者注：这是片段 1 中，setTimeout 和 setImmediate 执行的顺序不固定的原因。*

我们之前谈过，setImmediate 会减少额外的检查。因此 setImmediate 会执行更快一些。它也放置在轮询阶段之后，因此来自于任何一个到来的请求 setImmediate 回调将会立即被执行。

为什么 setImmediate 会被立即调用?

setImmediate 和 process.nextTick() 都命名错了。所以功能上，setImmediate 在下一个 tick 执行，nextTick 是马上执行的。😛

Javascript 代码能被阻塞吗?

像我们已经谈过的，nextTickQueue 没有回调执行的限制。因此如果你递归地执行 process.nextTick()，你的程序可能就永远在事件循环中出不来，无论你在其它阶段有什么。

如果我再 exit callback 阶段调用 setTimeout 会怎么样？

它可能会初始化计时器，但回调可能永远不会被调用。因为如果 node 在 exit callback 阶段，它已经跳出事件循环了。因此没有回去执行。

一些短的结论

- 事件循环没有工作栈
- 事件循环不在一个单独的线程里面，JavaScript 的执行也不是像从队列中弹出一个回调执行那么简单。
- setImmediate 没有将回调推入到工作队列的头部，有一个专门的阶段和队列。
- setImmediate 在下一个循环执行，nextTick 实际上是马上执行。
- 当心，如果递归调用的话，nextTickQueue 可能会阻塞你的 node 代码。